The LCG Dictionary and POOL

Dirk Duellmann

Reflection Data

What types of reflection information need to be available at runtime

- logical description of classes
  - data member names and types
  - method names signature

  => compiler/platform independent
- physical layout of classes
  - data member offsets and sizes
  - total class size (and sub-class offsets)

  => compiler & platform dependent
- dynamic (in core) information
  - function pointers for methods (including con-/destructors)

  => application/process dependent

Dictionary & Reflection

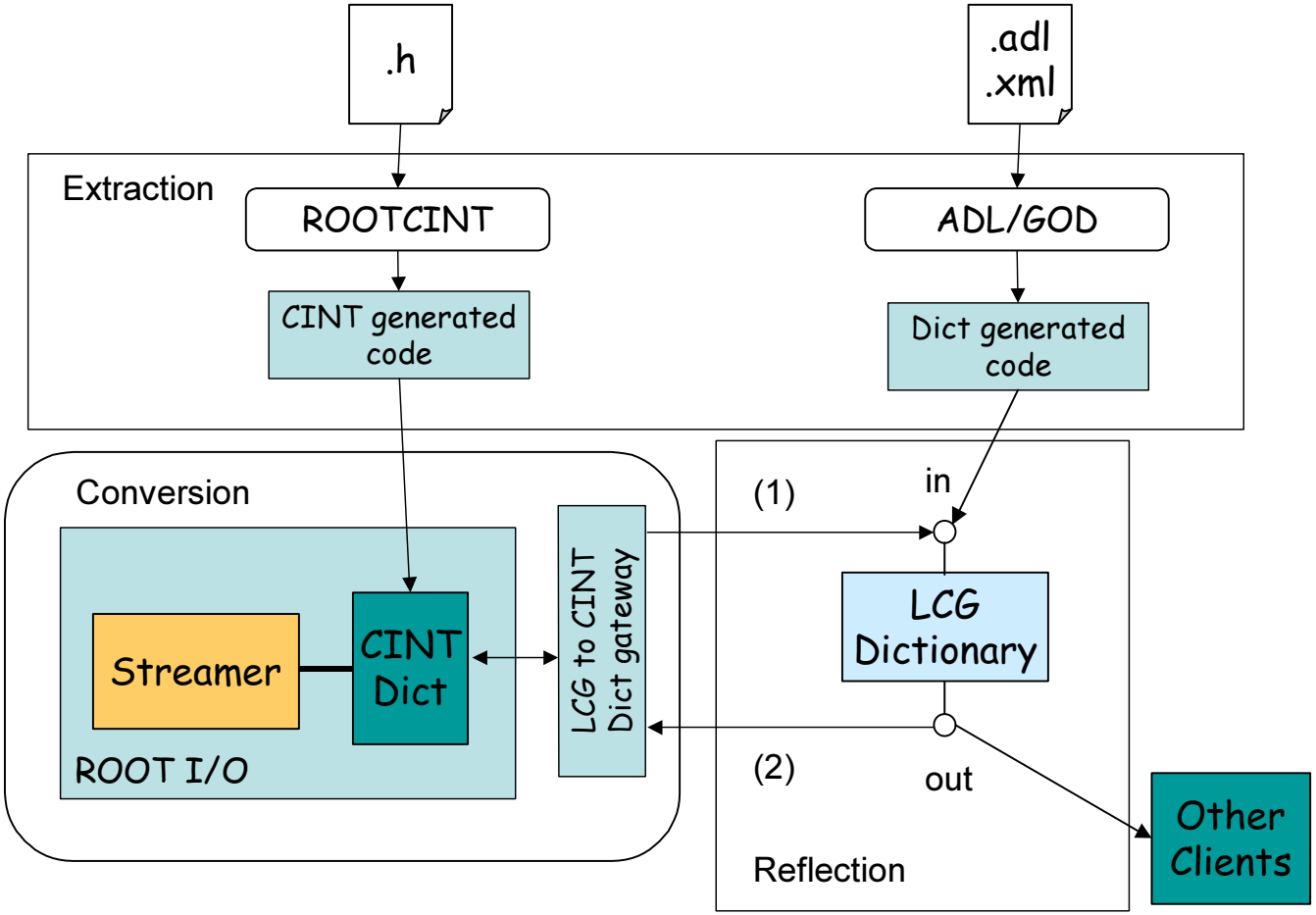
- Dictionary Components or Reflection Clients – Just a terminology proposal
  - Reflection
    - The component describing at runtime the transient data inheritance hierarchy and allowing runtime data & method access
    - Stop-gap for missing functionality of C++ as implementation language
    - Language implementation often come with build-in support (Python, Java, C#, SQL and of course Cint)
  - {Object} Conversion
    - Component which converts objects between a transient and a persistent representation
    - Client of the Reflection component
  - Extraction
    - Component which obtains reflection data
      - by parsing C++ files, or
      - during C++ code generation from meta model languages like ADL or XML, or
      - debug information or …
    - Client of the Reflection component which fills it with information
  - Language adapters (eg scripting)
    - component which allows to access C++ objects from other languages
    - not the scope of POOL, but closely related to the Reflection component

POOL and the LCG Dictionary

In POOL we try to keep the various dictionary components separate

- Why : POOL is supposed to be storage technology independent
  - POOL client code should be portable to other persistency back ends with minimal impact
  - We expect to support several Conversion and Extraction components
    - usually they are dependent on the persistency technology (conversion) or on the experiment using them (extraction)
    - existing implementations (eg from ROOT I/O) are re-used for implementation but directly exposing them would break technology independence
- LCG controlled Reflection API
  - The Reflection component plays a central role and is visible to POOL user (eg experiment framework implementers)
  - Need to define a stable API
    - which can be re-implemented for any backend storage
    - current LCG Reflection package was extracted from GAUDI
  - Need to connect this API to each technology dependent Conversion and Extraction components
    - Logically the content of the LCG and the technology dependent dictionary are kept in sync

Dictionary: Reflection / Extraction / Conversion
(adapted from Pere's original slide)



Persistent Reflection / "Dictionary"

Similar but not identical role to description of transient objects

- Only a subset of transient reflection info is stored or even makes sense on disk (basically the first two categories of the above)
- Several descriptions for a given class need to be kept to support schema evolution
  - the transient side (C++ code) can only cope with one C++ class layout per application
  - the disk may contain many
- Only both the persistent and transient reflection can steer the conversion to the C++ class expected by the current application

- POOL so far does not directly use or expose an abstract interface to the ROOT I/O disk representation
  - This information is used by the ROOT I/O conversion mechanism
  - This is a pragmatic divergence from the RTAG proposal to be able to integrate ROOT I/O without having to reimplement the ROOT conversion infrastructure
- The current POOL conversion service uses a technology dependent implementation
  - which is configured from a technology indepedent Reflection component

Longer term Plans : C++0x

- Potential good news :
  - IF C++0x comes with a standard for XTI and XPR
    - simplify the reflection info extraction
      - a gcc based extraction prototype seems to exist already
    - standardize the API to access c++ reflection info
      - a prototype interface/library seems to exist
- But we'll still have other reflection component around
  - conversion (one per storage technology)
    - ROOT team is very interested and may evolve towards xti
  - languages: python, java, C#, web services, sql (one per vendor)
  - can we assume all of those will interface with xti? by when?
- Even assuming a successful standardization of XTI for quite some time LCG will need to maintain a stable interface on its own